Concurrent deques

This document contains descriptions, arguments for correctness and ideas for improvement of the two concurrent deque implementations found at `https://github.com/arnegebert/deques`. A variation of this text was also submitted for evaluation in the Concurrent Algorithms and Data Structures take home exam at Oxford University in Michaelmas term 2019.

Lock-based deque

The implementation uses a doubly-linked list that always contains two dummy nodes *head* and *tail*. The elements of the abstract deque are all the values of nodes reachable from a dummy node *head* following the next-references, excluding the last node which is always the dummy node *tail*. The main idea of the implementation is for an operation to lock all shared nodes it needs access to.
Generally, the deque operations proceed in four steps:

1. lock the nodes that will be affected by this operation
2. validate that you locked the correct nodes (since you can't atomically acquire multiple locks)
3. manipulate pointers
4. release the locks

This gives rise to the invariant that pointers of unlocked nodes pointing to unlocked nodes are always up-to-date. Keeping this invariant in mind, the correctness of the implementation depends on 1) that the affected nodes are locked (justified below) and 2) correctly validating that the intended nodes were locked (justified in the description of the operations).

Justification for which nodes are locked: a remove of node x, manipulates the pointers of x.prev and x.next. The pointers of x should now longer be updated as well. As a result, removes respectively lock the first, second and third, or third last, second-to-last an last node (including the dummy nodes). An insert of node y between nodes x and z, manipulates the pointers of x, y and z. However, since y is not yet visible to the other threads, inserts only need to respectively lock the first and second or second-to-last and last node (including the dummy nodes). Deadlocks are avoided by letting all operations acquire locks in the same order ('from left to right', if *head* is left, proof follows below).

Description of operations: It is assumed that the code is open in parallel while this description is being read. To validating that n subsequent nodes are still subsequent after locking, we need to check that no deque operation occured on them between reading and locking. To check against removes, we validate for every locked node x that x is still reachable from its neighbours. This isn't necessary for the dummy nodes. To check against inserts, we validate for every locked node x that its pointer to locked neighbours did not change.

addLast saves *tail*.prev and *tail* into the values *prev* and *next* (order of reading doesn't matter) and then attempts to acquire locks on *prev* and *next* (order of locking matters). After locking, we need to validate that 1) no new node is between *prev* and *next* ('prev.next==next') and that 2) *prev* was not deleted ('next.prev==prev'). If validation fails, we release the locks and retry. If validation succeeds, we have locks on the second-to-last and last node of the deque, update pointers to insert the new node and release the locks (order of releasing doesn't matter).
addFirst proceeds completely analogue. The only difference worth pointing out is that the validation check only contains one expression now. This is because *prev* is now the non-deleteable dummy node *head* and the checks 1) whether no new node is between *prev* and *next*, and 2) that *prev* (instead of *next* compared to *addLast*) wasn't deleted, are both covered by 'prev.next==next'.

removeLast reads the values *tail*, *toRmv* and *prev*, and aborts immediately if the list has only 2 values. We then lock *tail*, *toRmv* and *prev* and validate that no new nodes were inserted between them ('prev.next==toRmv&&toRmv.next== and that none of the three nodes were deleted ('toRmv.prev==prev&&next.prev==toRmv'). If validation fails, we retry. If it succeeds, we update the pointers to remove *toRmv*, release the locks and return *toRmv*'s value.
Again, *removeFirst* proceeds completely analogue with an interesting observation being that we don't need to validate that 'toRmv.prev==head' if we already know that 'head.next==toRmv'.

The linearization points (LPs) of successful adds and removes are when the last lock they need is acquired. After this moment in time the add / remove can no longer be stopped because all relevant nodes are locked until the operation is complete.

A remove fails when *head*.next points to *tail* / *tail*.prev points to *head* at the time *toRmv* is being read and thus these reads are the LPs of failed removes.

Claim: Pointers of unlocked nodes pointing to unlocked nodes are always up-to-date.
Proof: The invariant obviously holds when the deque is empty. In the general case, we know that an operation will only manipulate the linked list once it acquired locks on the to-be-manipulated nodes. Since, according to the invariant, the node's pointers were correct before the manipulation, they will be correct after the manipulation (assuming the manipulation is done correctly). Since locked nodes are (or just were) manipulated, we can only make weaker guarantees for them (which are not relevant for our implementation).

Claim: This implementation is deadlock-free.
Proof through contradiction: assume this implementation is not deadlock-free. Then assume there is an operation a that locked node A and an operation b that locked B. a wants to acquire the lock on B and b wants to acquire the lock on A. However, all operations acquire locks on nodes they need in the same order. Therefore, w.l.o.g. if b requires a lock on A, it would have acquired this lock before acquiring the lock on B. Contradiction $\Rightarrow\Leftarrow$. Thus, the claim holds. □

Two inserts at different ends will only interfere if the deque contains less than 2 elements, one insert and one delete only if the deque contains less than 3 elements and two deletes only if the deque contains less than 4 elements. Concurrent operations at same ends always interfere, independent of deque size.

In highly concurrent systems, our implementation could be more efficient with backups locks reducing contention. By far the biggest efficiency improvement would be making the implementation lock-less (as done below)). This can be done through reintroducing marked (logically deleted) nodes, only requiring eventual correctness for .prev-pointers and helping between threads. This could also be done partially, e.g. reintroducing marked nodes would mean we only need to lock 2 (instead of 3) nodes in removes because we no longer need to lock the to-be-removed node. The implementation could likely also work with marked nodes and finer-grained pointer-wise instead of node-wise locks.

Lock-free deque

Definitions:
predecessor of a node n := unmarked, unique node whose .next-pointer points to n.
pred(n) := predecessor of a node n
successor of a node n := first unmarked node earliest reachable via n.next-path
succ(n) := successor of a node n

The implementation uses a doubly-linked list that always contains two dummy nodes *head* and *tail*. The elements of the abstract deque are all the values of unmarked nodes reachable from a dummy node *head* following the next-references, excluding the last node which is always the dummy node *tail*. Importantly, the .prev-pointer are not always up-to-date as we cannot update both the .next- and the .prev-pointer atomically). Instead, temporarily we only require for each node $n \neq head$ that we can follow the next-references from node n.prev to get back to node n (invariant (a) below).

The operations are generally of the form:

1. if the operation is a remove, mark the to-be-removed node.
2. update the .next-pointer.
3. update the .prev-pointer.

The most important invariants are:

(a) .prev-pointers are only eventually correct and temporarily we only require for each node $n \neq head$ that we can follow the next-references from node n.prev to get back to node n.

(b) .next-pointers of unmarked nodes never skip over unmarked nodes.

I first introduce multiple help operations used by the four basic operations, before introducing the basic operations themselves. The help operations contain the core logic of the implementation and correctness of the four basic operations is straightforward given the correctness of the help operations. Thus, the help operations are justified a lot more extensively than the basic operations themselves.

updatePredGuess takes a node n and the current guess for n's predecessor *predGuess*, and updates *predGuess* such that it is 'closer' to pred(n). It returns the updated guess for *predGuess*.

It operates in the following way: If *predGuess* is marked, it can't be the predecessor and we backtrack to *predGuess*.prev. It is safe to backtrack here because 1) in the worst-case we will backtrack (through possibly several marked nodes) to land at *tail* (follows from (a)) and 2) this won't lead to a deadloop where we always get to the marked node and then backtrack because the marked node 'on the way' will eventually be skipped over.
It would not be safe to advance to *predGuess*.next because we might skip the predecessor of node (since pointers of marked nodes can be out of date). It would not be safe to backtrack to *n*.prev because this leads to an endless loop if *n*.prev is marked. Backtracking to *tail* (the only other node that is always safe) would be less efficient.
If *predGuess* isn't marked, it could be the predecessor and we check whether it is ('predGuess.next==node'). If it is, we can return *predGuess* unchanged. If it isn't, we advance to *predGuess*.next. This is safe (i.e. never leads us to skipping over the predecessor) because of invariant (b). Thus *updatePredGuess* will always return a correctly updated guess for the predecessor of n. *updatePredGuess* is perhaps the most important function of the implementation.

updateAndGetPrev takes a node *n*, updates the potentially-outdated *n*.prev and returns the updated *n*.prev (or aborts and returns the currently guessed predecessor *predGuess* if *n* becomes marked). The initial guess for *n*'s predecessor *predGuess* is *n*.prev (safe since it is guaranteed to be 'left' of *n*) and *updateAndGetPrev* iterates in the following way:
- if *n* is marked return *prev*. This is necessary to prevent NullPointerExceptions when *n* is deleted and no longer reachable, and *updatePredGuess* advances *predGuess* to *tail*.next (which is null). This is correct because it is not necessary to update the .prev-pointer of a deleted node.
- update the current guess for *predGuess* using the function *updatePredGuess*. Correctness follows from the correctness of *updatePredGuess*.
- check if *prev* is the actual predecessor via 'prev.next==node' and if it is, attempt to update *n*.prev via a CAS. If the CAS succeeds, return *n*.prev. Else continue looping. This is correct because the CAS will fail, if the predecessor has changed due to *predGuess* or *n* beeing marked between calling *updatePredGuess* and the CAS or between 'if (node.mark)' and the CAS. The CAS will still succeed if the predecessor changes due to inserts between '*predGuess*.next==node' and the CAS. However, this is not an issue because we know that *predGuess* is not marked and we allow temporarily inconsistent .prev-pointers (see invariant (a)). Thus *updateAndGetPrev* will only update to and return an allowed predecessor.

skipOverNode takes a marked node *n* whose predecessor should be updated to 'skip over' *n*. It only terminates once *n* is no longer reachable by the .next-path starting at *head* and ending at *tail*.
predGuess refers to the current guess for the predecessor of *n* and *next* refers to *n*'s successor. *skipOverNode* loops in the following way:
- if *predGuess* and *next* point to the same node, *n* has been skipped over and *skipOverNode* can terminate. For correctness see Lemma 1.
- if *next* is marked, advance it and restart loop. Correctness see Lemma 2.
- update *predGuess* with *updatePredGuess*. Correctness follows from the correctness of *updatePredGuess*.
- check if *predGuess* is the actual predecessor via 'predGuess.next==node' and if it is, attempt to update *predGuess*.next to succ(n). This is correct because the CAS will fail if pred(n) or succ(n) changes: If *predGuess* or *next* is marked between loading *updatePredGuess* and the CAS or 'if (node.mark)' and the CAS, the CAS fails. If a new node gets inserted between *prev* and *next*, the CAS fails. If the CAS succeeds, *n* has been skipped over and *skipOverNode* can terminate.

Lemma 1: If and only if a node *n* is skipped over, the condition 'prev==next' will be true.
Proof:
(α) *prev* will never overtake *next* before they're pointing to the same node (through 'skipping over' *next*). This follows from the facts that *next* is advanced before *prev*, *next* is advanced when it's marked and invariant (b).
(β) If *n* is no longer reachable by the .next-path starting at *head* and ending at *tail*, then *prev* and *next* will point to the same node. From the correctness of *updatePredGuess*, it follows that *predGuess* will continuously advance through all unmarked nodes in the list from left (not necessarily starting from *head*) to right to find pred(n). Since pred(n) is no longer reachable, but *updatePredGuess* continues advancing from left to right, it follows from (α) that *prev* and *next* will eventually point to same node.
(γ) If a node *n* is not skipped over, 'prev==next' will never be true. This follows from the correctness of *updatePredGuess* and the fact that *skipOverNode* will terminate if '*predGuess*.next==node' and the CAS succeeds.
The claim follows from (α), (β) and (γ). □

Lemma 2: As long as a marked node *n* is not skipped over, *next* will be updated until it points to succ(n).
Proof through considering all cases where succ(n) may change: succ(n) may change when either the previous succ(n) is marked or a node gets inserted between pred(n) and succ(n). When a node *n2* gets inserted between

pred(n) and succ(n), pred(n).next will change from n to $n2$ as part of the insertion. Therefore, n is skipped over and succ(n) is no longer required to be correct (according to the claim). When the previous succ(n) is marked, *next* is advanced through .next-pointers until an unmarked node is detected. An unmarked node inserted between pred(n) and succ(n) could be skipped by this chain of .next-pointers, but in this case *next* no longers needs to point to succ(n). Otherwise, the chain of .next-pointers finds the next unmarked node after n, which per definition is succ(n). Thus the claim holds. □

addFirst: After initializing and setting the pointers of the new node *node*, we repeatedly attempt to update the .next-pointer of *head* to *node* through a CAS operaton. If the CAS operation fails, the saved *next* is no longer up-to-date and we reload it. When the CAS succeeds, we update the .prev-pointer of *node*.next via *updateAndGetPrev*. *addFirst* is correct because the CAS only succeeds when *next* is still succ(*head*) and because *updateAndGetPrev* is correct.
addLast generally operates in the same manner, however, we need to read pred(*tail*) with the operation *updateAndGetPrev* because *tail*.prev might be outdated (see invariant (a)). *addLast* is correct because the CAS only succeeds when *prev* is still pred(*tail*) and because *updateAndGetPrev* is correct.
The linearizability points of *addFirst* and *addLast* are the successful CAS operations. From this time onward the new node is in the abstract deque and before that concurrent modifications may have prevented adding of the node.

removeFirst immediately returns if it detects that the list only contains the dummy nodes *head* and tail. Else it checks, whether the successor of *head* is marked (this happens when another remove already 'claimed' that node by marking it, but didn't update the .next-pointer of *head* yet) and helps with updating the .next-pointer of *head*. If neither of the above is the case, it attempts to mark *toRmv* to 'claim' it. If the CAS fails, it goes through the previous steps again. If the CAS succeeds, it updates the .next-pointer of the predecessor of *toRmv* (which does not need to be *head*) and the .prev-pointer of the successor of *toRmv* to physically remove it, and returns the value of *toRmv*. *removeFirst* is correct because the CAS only succeeds when we are the first to mark succ(*head*) and because *skipOverNode* and *updateAndGetPrev* are correct.
Again, *removeLast* operates in a similar manner to *removeFirst* and I will only go over the interesting parts. *toRmv* is now loaded through the *updateAndGetPrev* function, which returns the predecessor of tail (after potentially updating *tail*.prev). This is necessary because *prev.tail* might be outdated (see invariant (a)). *removeLast* is correct because the CAS only succeeds when we are the first to mark pred(tail) and because *skipOverNode* and *updateAndGetPrev* are correct.

The linearization points of successful removes is the read of *toRmv* in the iteration the remove succeeds. Notice that the linearization points can't be the successful CAS operations because the CAS operations do not fail if a node is inserted between the read of *toRmv* and the CAS operations. A sequence of operations where an insert overtakes a dequeue at the same end would then lead to an incorrect linearization.
A *removeFirst* fails if *head* points to *tail* at the moment in time when *head*.next. Similarly, a *removeLast* fails if *head* points to *tail* at the moment in time when *prev*.next is read. Therefore, the linearization points of failed removes is the read of *toRmv*.

Similar to backup locks in the lock-based version, spinning / backups after failed CAS attempts could reduce contention and increase efficiency in highly concurrent systems.
updatePredGuess frequently has to backtrack because a node on 'its path' is marked and it has to wait for the marked node to be removed by another thread. Allowing *updatePredGuess* to 'help itself' by calling *skipOverNode* on the marked node could reduce backtracking and waiting on other threads a lot. This could be made even more efficient, if *updatePredGuess* checks whether *predGuess*.next is marked before advancing on to it, and if it is, give *skipOverNode predGuess* from *updatePredGuess* as guess for pred(*predGuess*.next) to avoid *skipOverNode* having to search for pred(*predGuess*.next). (This would mean that *skipOverNode* would need to be extended to receive an optional *initialPredGuess* argument)

The final implementations were tested with the included linearizability tester for approximately 8 million iterations with 4 workers and 20 operations per worker. The probability of a remove was 60% (30% for rmvFirst/Last) and the probability of an insert was 40% (20% for insertFirst/Last). These probabilities were chosen so the deque was most often in a critical state with only very few elements. The LockFree version was additionally tested with randomly paused threads to reliably simulate threads being overtaken.